Chapter I.
Introduction to Compilers (Objectives)

a.hettab@centre-univ-mila.dz

What is a compiler?

- A **compiler** is a **translator** that transforms a program written in a language **L1** into another program written in a language **L2**.
- The language L1, in which the original program is written, is the **source language**:
  - e.g., **C, C++, Pascal …**
- The language **L2**, generated from language **L1**, is the **target language** (or **object language**) :
  - e.g., **EXE,…**
- In practice, compilation often stops at an **intermediate language** (assembly or even the language of an abstract machine).

What is a program?

- A program is a set of operations that describe how to produce results from inputs.
- The compiler rejects programs that contain errors (static errors);
- otherwise, it builds a new program (the object program) that the machine can execute on different inputs.
- The execution of the object code on a particular input may fail to terminate or fail to produce a result (dynamic error).

Example of compilation

```
1  int squareSum (int a, int b)
2  {
3    return a*a+b*b;
4  }
```

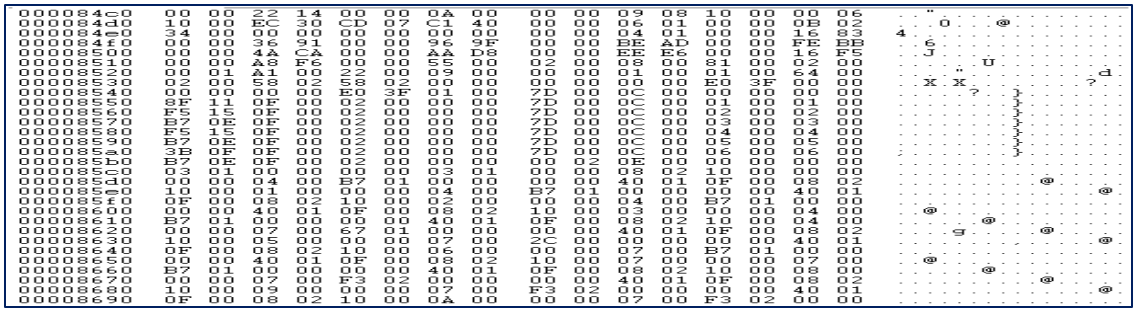
Source program: C language

Compiler gcc

```
1  <squareSum>:
2  0:  0f  af  ff      imul  %edi,  %edi
3  3:  0f  af  f6      imul  %esi,  %esi
4  6:  8d  04  3e      lea   (%rsi, %rdi,1), %eax
5  9:  c3      retq
```

Generated Assembler Program

Assembler

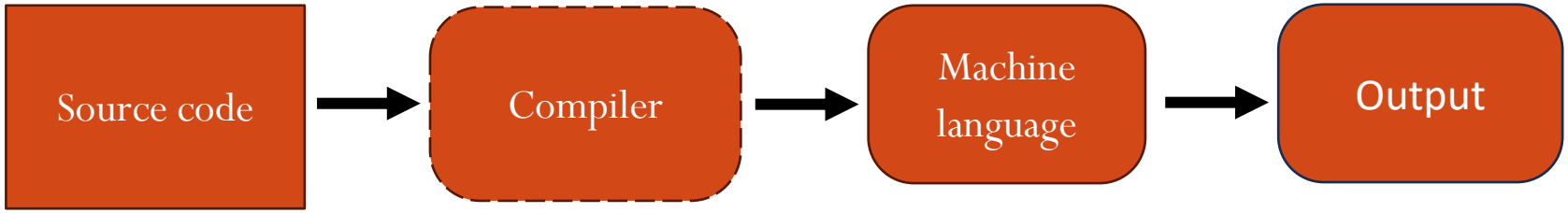


Generated machine code

Compiler vs Interpreter

Compiler

- A compiler takes the entire program and converts it into object code, which is usually stored in a file. The object code is also referred to as binary code and can be executed directly by the machine after linking.
- Example: **C**, **C++**, **Pascal …**



Compiler vs Interpreter

Interpreter

- An **interpreter** directly executes instructions written in a programming language one after another without converting them into object code or machine code.
- Example: **BASIC, Perl, Python, Ruby, PHP ….**

Compiler vs Interpreter

Advantage and disadvantage

	Advantage	Disadvantage
Interpreter	Simple development process (especially debugging))	Inefficient translation process and slow execution speed
Compiler	Delivers the complete ready-to-use and executable machine code to the processor	Any modification of the code requires a new translation (error correction, software extension, etc.)

Compiler vs Interpreter

just-in-time compiler (JIT compiler)

- **Just-in-time compilation** is a hybrid solution that translates the program's code during execution, similar to an interpreter. It combines the two previous methods to take advantage of their benefits. In this way, the **high execution speed** (enabled by the compiler) is complemented **by a simplified development** process (enabled by the Interpreter).
- **Example**: **JAVA…**

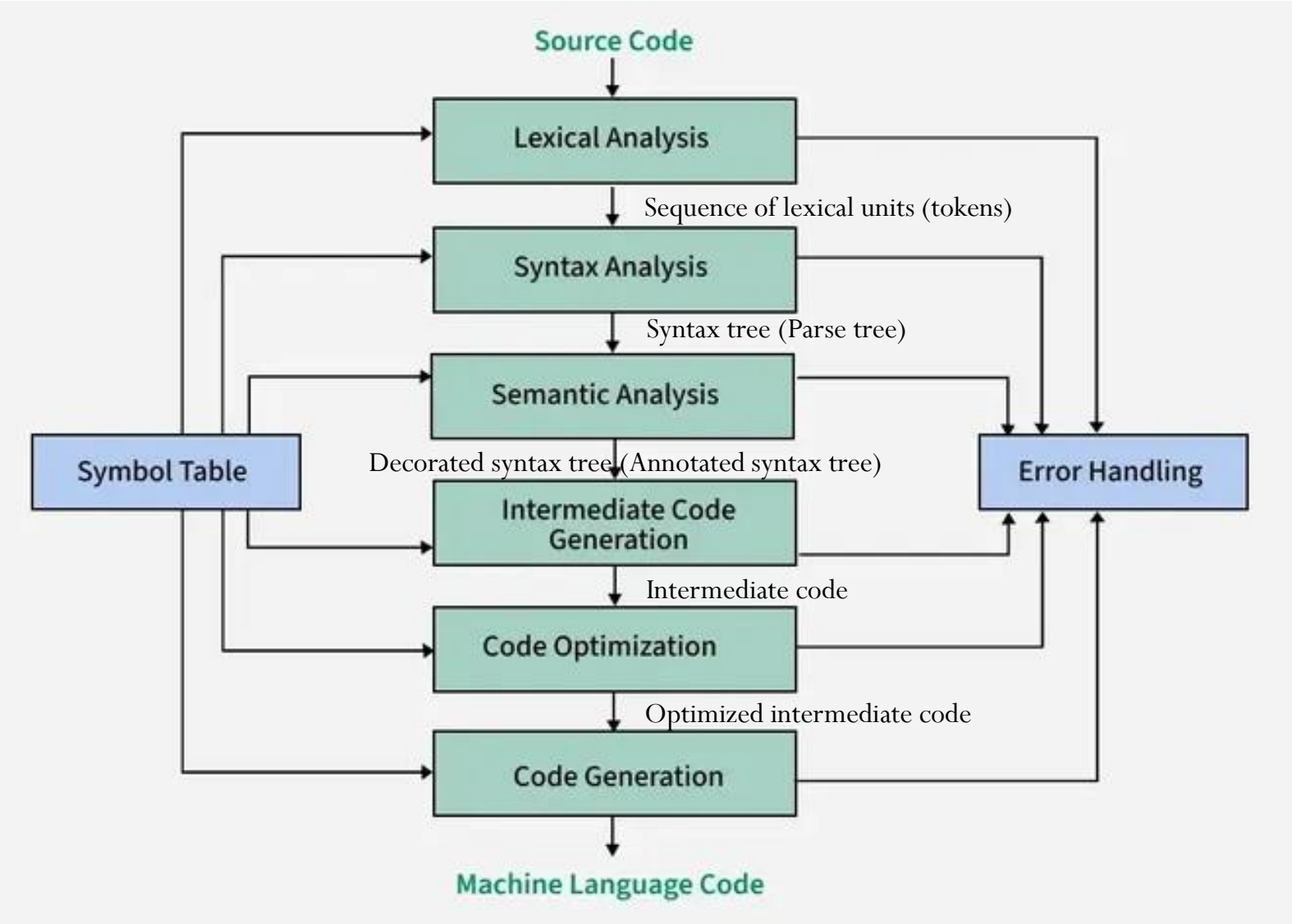
What is expected of a compiler? (1)

- **Error detection**: A compiler must be able to detect static errors such as :
  - Malformed identifiers,
  - Unclosed comments,
  - Incorrect syntactic constructs,
  - Undeclared identifiers,
  - Ill-typed expressions, **e.g., if 3 then "toto" else 4.5,**
  - Uninstantiated references.

What is expected of a compiler? (2)

- **Efficiency:** A compiler should be as fast as possible.
- **Correctness:** The compiled program must represent the same computation as the original program.
- **Modality:** Use of separate compilation during large-scale development.

Structure of a compiler



Lexical analysis

- Lexical analysis consists of transforming code in textual form into a sequence of **tokens (lexical units)**, thereby separating keywords, variables, integers, etc.
- Comments and the spaces separating the characters that form the lexical units are removed during this phase.
- The lexical analyzer assigns each lexical unit a code specifying its type and a value (a pointer to the symbol table).

Example

for i :=1 to vmax do a :=a+i;

We can extract the following sequence of tokens:

for : keyword
i : identifier
:= : assignment
1 : integer
to : keyword
vmax : identifier
do : keyword
a : identifier
:= : assignment
a : identifier
+ : arithmetic operator
i : identifier
; : separator

Symbol table

- A symbol table is a centralization of the information associated with the identifiers of a computer program.
- In a symbol table, you can find information such as the type, memory location, etc.
- Generally, the table is created dynamically. An initial portion is created at the beginning of compilation, and then, it is completed as needed.
- The first time a symbol is encountered (according to the language's visibility rules), an entry is created in the table.

Symbol table

N°	lexical unit (tokan)	Type of the lexical unit (tokan unit)
10	for	keyword
11	to	keyword
12	do	keyword
13	;	separator
…	……	……
100	:=	assignment
101	+	arithmetic operator
…	….	….
1000	i	identifier
1001	a	identifier
1002	vmax	identifier
….	…..	
5000	1	integer

Then, the previous statement can be expressed as follows: 10, 1000, 100, 5000, 11, 1002, 12, 1001, 100, 1001, 101, 1000, 13

Syntax analysis (Parsing)

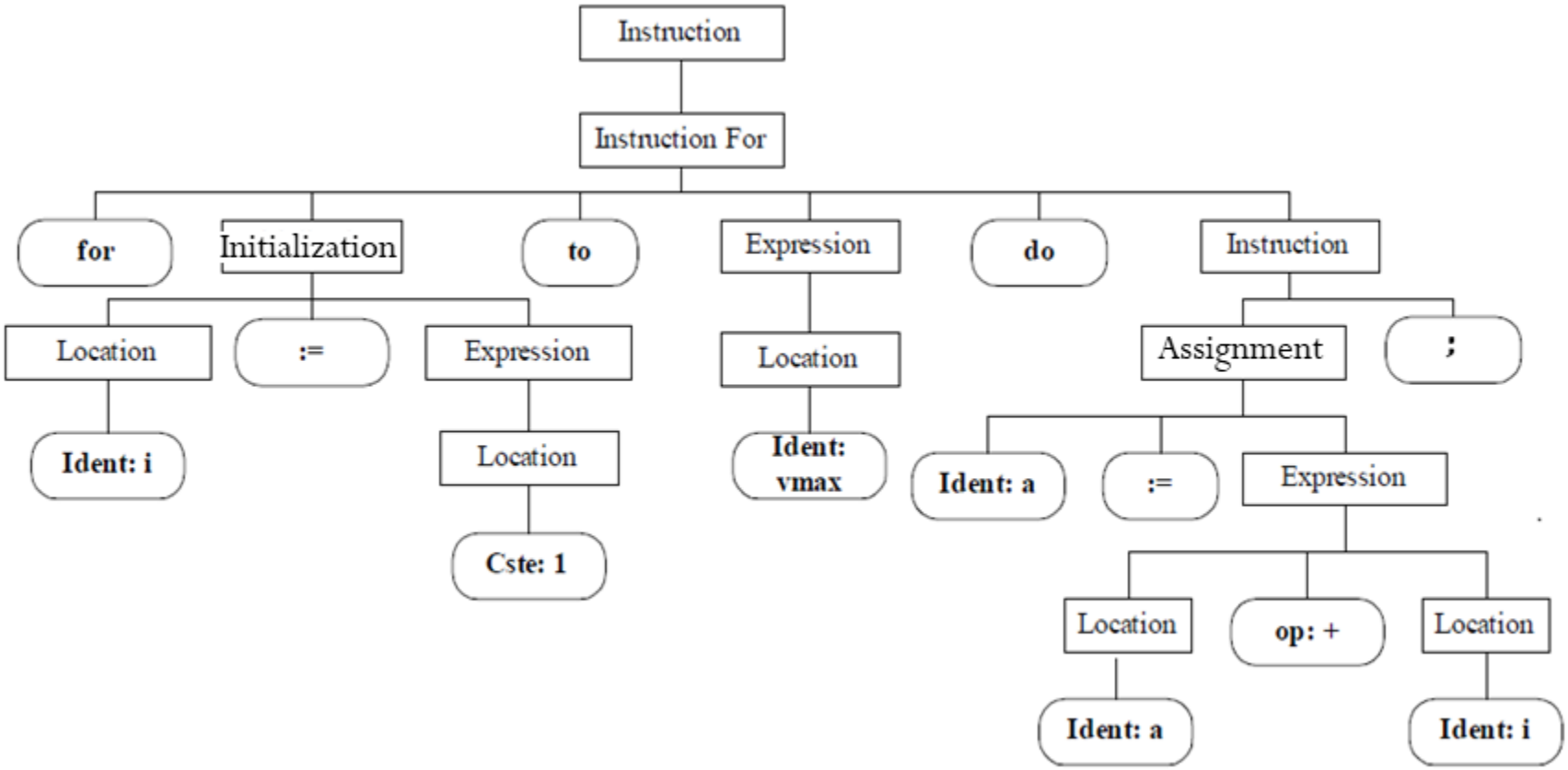
- This phase consists of grouping the lexical units of the source program into grammatical structures (i.e., verifying whether a program is correctly written according to the grammar that specifies the syntactic structure of the language).
  In general, this analysis is represented by a tree.

Example

- Let us consider the following grammar for the for loop:
- **<Instruction> → <Assignment> ; | <For instruction>**
- **<Assignment> → ident := <Expression>**
- **<For instruction> → for <Initialization> to <Expression> do <Instruction>**
- **<Initialization> → <Location> := <Expression> | <Location>**
- **<Location> → ident | const**
- **<Expression> → <Location> | <Location> op <Location>**

Syntax tree (Parse tree)

- After syntax analysis, we obtain the following tree for the statement: **for i :=1 to vmax do a :=a+i;**



Semantic analysis

- Semantic analysis is used to specify the nature of the computations represented by a program by verifying that the operands of each operator comply with the specifications of the source language.
- In general, this analysis is represented by a decorated syntax tree

Example:

- It is necessary to verify that the variable 'i' indeed has the type 'integer,' and that the variable 'a' is indeed a number. This operation is performed by traversing the syntax tree and checking at each level that the operations are correct.

Intermediate code generation

- After the semantic analysis phases, some compilers produce an intermediate representation, a kind of code for an abstract machine.
- The "**three-address code**" intermediate form is widely used.
- Using intermediate code offers several advantages:
  - it is relatively easy to translate intermediate code into object code;
  - the intermediate representation allows optimizations that are independent of the target language (object code).

Example: **Java** uses a specific intermediate form: **bytecode**. Bytecode can be executed on any platform using the **Java Virtual Machine (JVM)**.

Intermediate code optimization (1)

- Optimization aims to improve the intermediate code in order to enhance its performance by reducing execution time or/and memory usage.
- Some optimization operations include :
  - **Loop-invariant code motion :**

    The goal is to extract from the loop body any parts of the code that are invariants (requiring only a single execution). This reduces the total number of executed instructions.

Example. Let the following intermediate code be:

```
for (int i = 0; i < n; i++) {
 x = y + z;
 a[i] = 6 * i + x * x; }
```

After moving the invariant code, we obtain:

```
x = y + z;
temp1 = x * x;
for (int i = 0; i < n; i++) {
 a[i] = 6 * i + temp1; }
```

Intermediate code optimization (2)

- Some optimization operations include :
  - **Dead code elimination :**

  The compiler can recognize parts of the code that are dead (the reason may be a programming error).

Example . Instructions that are unreachable due to jump statements are considered dead code and can be eliminated :

```
if 1 <> 0 goto label
i := a[k]
k := k + 1
label:
```

Intermediate code optimization (3)

- Some optimization operations include :
  - **Common subexpression elimination :**

  When the value of an expression is computed in multiple places in the program, the optimizer stores the result of this expression and reuses it instead of recalculating it.

Example . Let the following intermediate code be :

t6 = 4 * i
x = a[t6]
t7 = 4 * i
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = 4 * j
a[t10] = x

After eliminating common subexpressions (and applying some other optimizations), we obtain:

t6 = 4 * i
x = a[t6]
t8 = 4 * j
t9 = a[t8]
a[t6] = t9
a[t8] = x

Object code generation

- This phase produces object code by :
  - Choosing memory locations for the data;
  - Selecting machine code to implement the intermediate code instructions;
  - Allocating registers.

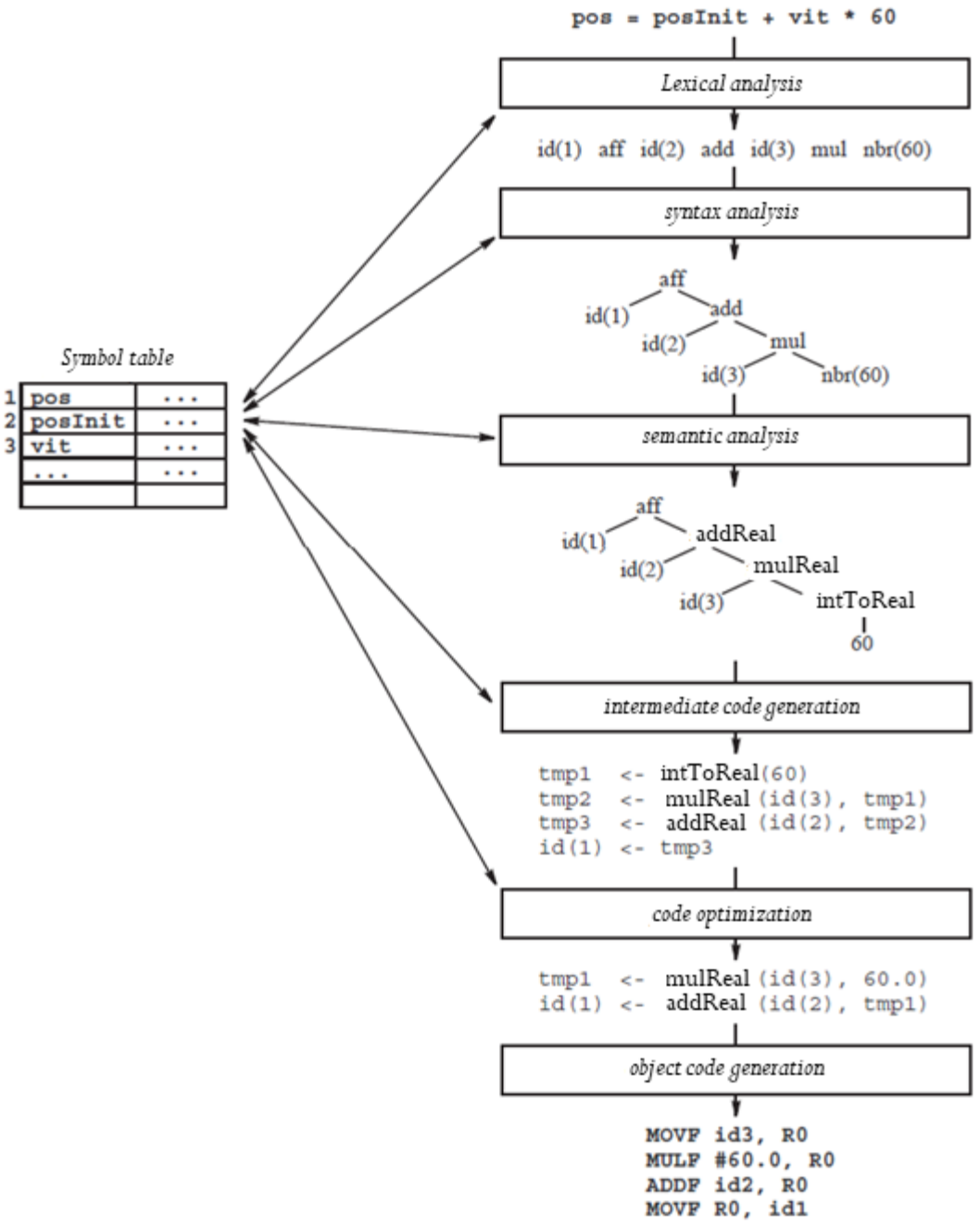
Example: Here is the following optimized intermediate code :

temp1 := 60.0 * id3
id1 := id2 + temp1

Here is the object code generated using registers R1 and R2 :

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Logical phases of compiling an instruction



References

- **Compilateurs Principes, techniques et outils**. Alfred V.Aho, Ravi Sethi, Jeffrey D.Ullman. Pearson. Paris France. 2007.